Cybersecurity Midterm Project Report

DSCI6672 – Spring 2020

Nancirose Piazza

Overview and Requirements

The requirement of the midterm project is as followed: to train a deep neural network, deploy the model through Amazon SagaMaker, and create a python script that takes in a PE as a parameter and classify as one of the following binary classes: as Benign or Malicious. The following report is partitioned into three subtasks on the complications and approach to creating a solution.

Task 1: Training

Despite the name of this task, the data must be preprocessed and vectorized to be fed into a deep neural network. Initially, I was unaware of the LIEF project and spent three days analyzing the data (ember-2017_2) through the dataset's published arxiv paper and converting to vectors. After becoming aware of the library, I immediately scrapped my vectorized data and installed their EMBER library.

The notebook was created on Google's Collaboratory service which led to a major constraint such as: Overloading RAM that caused session crashes. Because of this, it was unwise to download the data to the runtime session every time the session timed out or crashed. Therefore, Google Drive was used to host the data files. However, that led to another constraint. Google's free tier for every user is 15GB. This is enough to host a single copy of the data, but not more than that. Therefore, my solution was to download the files to the runtime session, install EMBER and vectorize the data. Then, mount my drive to copy over the .dat files. Because the files are referenced through numpy's memmap, the data

does not have to be loaded into the RAM at once; however, this would lead to slower processing of data to a machine learning model.

After constructing the full training data, full test data and their labels, the training data had to be separated from the valid classes and the unlabeled classes (samples labeled with -1). By using a mask however, it loaded all the data into the RAM, however it was also a longer process. Therefore, by passing commands through the notebook to remove the saved data from the Google Drive, space was available to save the newly almost useable training and testing data. It should be noted that by being able to load the full dataset into the RAM, the model's training takes less time.

The last preprocessing step for the data is to standardize or normalize it. This is needed to allow the neural network to converge, if not at least faster. The chosen normalizer was StandardScaler from the sklearn library. Because the data was too large to fit to the scaler instance at once, partial_fit was used on partitions of the training data. Finally, by creating two scaler instances for test and train data, we transform the data and rewrite it in memory to preserve space. Once again, the files were written to Google Drive after clearing previously saved data. The data was ready to be given to a neural network.

The considered architectures for the neural network were between two general concepts: A model with Conv1D as its first layer after input layer or a typical dense layer. It should be noted that because of the upgrade in Tensorflow between 1.x and 2.x, there is some required reshaping if using a Conv1D layer, eg. Data in the format of (v[1],v[2],v[3]) where there is 2 axes, (an array of an array of numbers) to 3 axes in the format ([v[1]],[v[2]],[v[3]]) (an array of tuples.) The selected architecture is as followed:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout (Dropout)            (None, 1, 2381)           0
_________________________________________________________________
dense (Dense)                (None, 1, 1500)           3573000
_________________________________________________________________
dropout_1 (Dropout)          (None, 1, 1500)           0
_________________________________________________________________
dense_1 (Dense)              (None, 1, 1)              1501
=================================================================
Total params: 3,574,501
Trainable params: 3,574,501
Non-trainable params: 0
```

It is recommended to add dropout layers to produce better generalization. The decision for the number of nodes in the dense layer is the size of approximately 2/3 of the number of features: 2381. The choice of activation functions were the following: relu for hidden layers, sigmoid for output layer, and Adam as the optimizer with the set learning rate of .01. There is a l1 regularizer set to the dense layer at .01 which is to prevent overfitting. The selected performance metrics were accuracy, AUC, and precision because false-alarm rate and detection rate are import to this domain. The wide architecture was recommended from the book, Deep Learning with Python by François Chollet.

For training, it was recommended to do small batches. For example, the model that showed the dataset did batches of 256 and MalConv that came with EMBER did batches of 100. Since Google offers free usage of their 12GB CPU, GPU and TPU, I opted for 256. Time variated depending on the RAM chip, but within half an hour, the model finished through one epoch, providing a promising 87-90% accuracy, 96% AUC, and 96% precision on training data and 98%,99%,98% on validation data respectively.

After saving the first preliminary round, the model was set to run for 30 epochs with EarlyStopping as a callback overnight to see if it'd produce better results. The model ran for

another 21 epochs before it stopped but despite longer training, the model seemed to get progressively worse as all the performance metrics continued to dip. When this happened, there were some altercations to the learning rate and batch size, but neither benefitted the model in this case. Those models were scrapped for the best preforming model that occurred after the first epoch.

The final evaluation was done on the test set, receiving 97.5%,98.5%,97.7% in accuracy, auc, and precision respectively. Additional metrics like f-score, 97.2%, and the confusion matrix were also calculated.

	pred_benign	pred_malicious
is_benign	97727	2273
is_malicious	3214	96786

Task 2: Deploy of Model on Cloud

The service used to deploy the model was Amazon's Web Services's SageMaker. Caution: Tensorflow 2.x is currently the most recent distribution of Tensorflow. AWS supports Tensorflow 2.0 for training and deploying housed TensorFlow Serving models from TensorFlow Estimators; however, to import an outside trained model, TensorFlowModel method is used to convert a TensorFlow Model into an estimator. The caveat is that the framework supports TensorFlow 1.x models, like 1.12 and 1.6 according to the tutorial and method documentation's default framework TensorFlow version. If one's intention is to deploy an outside model through SageMaker's Endpoint, it would be wise to assure their TensorFlow version is compatible with the TensorFlowModel method. In the tutorial resource on the deployment of a model through SageMaker, TensorFlow

1.12 is compatible. Google Collaboratory's current 1.x version is 1.5, but I could not deploy a functional model hosted by the Endpoint; therefore, downgrading to Tensorflow 1.12, recreating the architecture and loading the weights to resave the model was necessary. After getting the model to successfully import, the rest of the tutorial provided was sufficient. Depending the Amazon Web Service account; for example, an AWS educate account stores their keys and password before launching the AWS console; however, cannot be found elsewhere. A normal user AWS account can create their passwords and keys through the IAM, Identity Access Management.

The creation of the endpoint took around 7 minutes. When invoking the endpoint from within the SagaMaker notebook, response time was optimal—less than half a second.

Task 3: Create a Client

Moving away from Google Collaboratory, I launched a virtual machine through VirtualBox, specifically the distribution Ubuntu 18.04 to create an environment to write the code. This was an optimal testing environment for packages. Only one necessary file was bundled and that was the pickle scaler file. Normally, a client shouldn't have this file; instead, this file should be hosted on the cloud as well with a Lambda function set to scale incoming data but said service was restricted. The python script was written through the nano text editor and majority of the imports are already on the client except for EMBER. The python script installs EMBER if it is not installed. Putty.exe was used as the deployment sample and returned benign. Note: The code included for the client side has the sensitive keys and passwords removed. The skeleton of the code was provided by the EMBER GitHub repository and adjusted for this model. The response time for invoking the

endpoint and returning data was around 4 seconds. That is incredibly long for Malware detection and therefore should not be recommended for general use. This encourages the concept of having light models distributed on the client for more optimal speeds.

Conclusion

More complications and delays of this project were due to version incompatibilities and limitations in the service's free tier than of the objective themselves. This project should be forward to implement after knowing some of these complications. I spent time learning what Lambda and other services of AWS because many resources used them; however, due to account restrictions, could not use them. I believe I have better understanding why all deployed systems are practically legacy, products would be too unreliable otherwise. I also found passion of listing package versions ahead of time, say for a tutorial, because it'd make it easier to read through documentations.

Bibliography

H. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models", in ArXiv e-prints. Apr. 2018

Chollet, François. *Deep Learning with Python* . : Manning, 2017.